2.0 CONCEPTUAL MODEL SPECIFICATION

2.0.1 OPERATIONAL CONCEPT

EADSIM is an analytic model of air and missile warfare used for scenarios ranging from few-on-few to many-on-many. It is unique in that each platform (such as a fighter aircraft) is individually modeled, as is the interaction among the platforms. It models the command and control (C2) decision processes and the communications among the platforms on a message-by-message basis. Intelligence gathering is explicitly modeled and the intelligence information used in both offensive and defensive operations.

2.0.2 TOP-LEVEL SOFTWARE DESIGN

EADSIM consists of four processes which run in parallel and exchange data and commands required for their coordination. The processes:

- Command, Control, Communications and Intelligence (C3I)
- Flight Processing
- Detection
- Propagation

contain all or portions of the various functionalities designated in the SMART Functional Area Template (FAT), e.g., the Sensor element of the FAT has control logic to direct sensors and determine their status in the C3I process while Signal to Noise Ratio computations are performed and detections are determined in the Detection process. The C3I process performs Command and Control (C2) decision modeling, message processing, track processing, and engagement and weapon modeling for all platforms in a scenario. The state of a platform's C2 logic determines what computations are performed and what actions are modeled in the other three processes. The Flight Processing process maintains and updates the movement and status of each platform in response to commands and information received from C3I. The Detection process models each sensor in the scenario and determines, based on information provided by C3I, whether a sensor is active and when it can detect a given target. The Propagation process (Prop) models communications connectivity which determines whether messages can be successfully transferred between two nodes in a network.

EADSIM can be run in various configurations consisting of combinations of the four processes. Figure 2.0-1 shows the possible combinations that can be used. Notice that Flight Processing is always required and that it is the only process that can be run by itself. The most commonly used configuration is the one assuming perfect connectivity.

Processes Run				Uses
FP	DET	PROP	C3I	
•	•	•	•	• Full Combat Analysis
•	•		•	• Perfect Connectivity
•				• Flight Paths Only
•	•			• Non-reactive Case Detections
•		•		• Non-reactive Case Connectivity
•			•	• Tactical Missile Launches, Bomber Engagements

FIGURE 2.0-1. EADSIM Runtime Configurations.

Process Logic Flow

EADSIM is both an event driven and a time stepped simulation. The C3I process is event driven, i.e., uses an event calendar, while the other three processes are time stepped. The time step, called the simulation interval, can be specified by the user as an integer number of seconds or has a default value of 3 seconds. Each of the time stepped processes is controlled by a main program which, after a series of initialization steps enters into a processing loop that updates its scenario time, reads incoming messages from the other processes, performs its local processing and constructs and sends outgoing messages to the other processes. The processing loop terminates when its simulation time exceeds the end of the scenario.

Figure 2.0-2 diagrams how messages are exchanged among the processes to maintain the proper sequence of execution between them. The key to understanding the diagram is the fact that the communications sockets are blocking sockets, i.e., a process trying to read a socket must wait there until all the data in the message it is reading has arrived. This is the mechanism by which the required sequencing between processes is maintained. The top line of processing in the figure, labeled FP, represents the main processing loop for Flight Processing. The second line, represents the processing performed by C3I process, and the remaining two lines represent the main processing loops for the Detection and the Propagation processes, respectively.

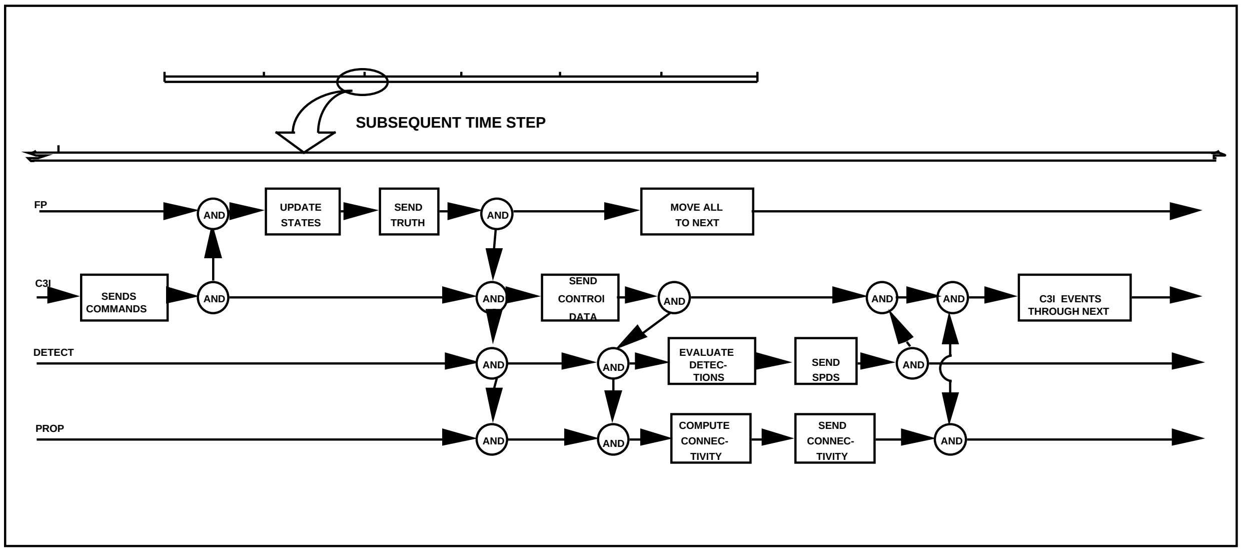


FIGURE 2.0-2. Execution Sequencing between Processes.

C3I sends the engagement commands, generated from the previous simulation interval, to the FP process which updates the state vectors of the platforms in the scenario, and uses the C3I commands to update the status of the platforms. The updated states and status information are then sent to the other three processes.

When C3I receives the updated platform data from FP, it sends sensor control information to Detection and jammer control information to both Detection and Propagation. Detection uses the platform states and status data in generating detections for the current simulation interval. The detections are placed in an array named the Sensor Periodic Detection Summary (SPDS). When all the detections for the current interval have been generated, the SPDS is sent to C3I. Propagation uses the C3I commands and the platform states and status from FP to determine network connectivities. The connectivities are sent to C3I, which uses the detections to perform track processing. The connectivity information determines whether a message can be received at its destination. Each of the time stepped processes advance their simulation clocks to the next simulation interval at the bottom of their processing loop and wait at the socket read for incoming messages. C3I advances its clock each time an event is taken from the calendar. After the interprocess communications (I/C) event is pulled from the calendar and the I/C function, C3I Data Transfer, is called C3I synchronizes with the other processes for the next simulation interval.

C3I Process Logic

Four major functions or functional classes comprise the description of the C3I process logic:

- the Simulation Executive
- C3I Initialization
- the generic processing performed in each ruleset
- the C3I Interprocess data transfer function.

The description follows Figures 2.0-3 through 2.0-7. Each figure is a logic flow diagram with numbered blocks that serve as keys in the discussion of the diagram.

The Simulation Executive

Figure 2.0-3 is a flow diagram of that part of the simulation executive which places the C3I functions into execution. The simulation executive consists of a set of scheduling utilities that schedule events on the event calendar and an Execute utility that retrieves each event from the calendar. Events can be scripted during scenario generation to occur at specified times or they can be scheduled in the course of processing other events. Each event contains a pointer to a function. The function performs the processing required to model the event. The Execute utility invokes the function when it retrieves the event. This can be thought of as a “ruleset engine” that allows the state of a ruleset to change in response to developing conditions as the scenario unfolds.

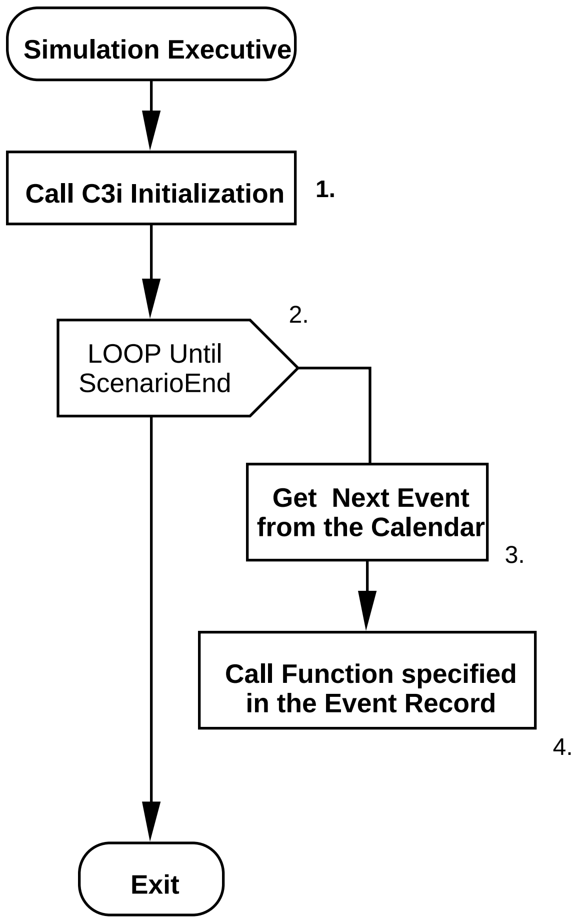


FIGURE 2.0-3. EADSIM Simulation Executive.

<u>Block 1</u>. The executive calls the initialization function, C3I Init, to read the scenario file and construct the runtime database of platforms, rulesets, sensors and jammers. When these initialization steps are complete, the first rulesets to be executed have been scheduled on the simulation calendar along with periodically executing utilities, such as C3I Data Transfer. The initialization function returns to the simulation executive.

<u>Block 2</u>. The simulation executive enters into a processing loop which continues to execute for the duration of the scenario.

<u>Blocks 3. and 4.</u> The simulation executive retrieves the next event to be executed from the calendar and extracts the pointer to the function that should be executed. The call to C3IDataTransfer is the first event placed on the simulation calendar in the simulation interval. This first call sets up the sockets with the other processes and provides the initial messages required to start C3I. The details of the data transfer function are discussed below. Subsequent call to this function are always scheduled to occur at the beginning of the next simulation interval. The remaining events on the calendar for this simulation interval are calls to ruleset phases that have been scheduled during the execution of previous events.

C3I Initialization Logic

The C3I initialization function, C3iInit, initializes the scenario data base and schedules events for the first scenario interval, as well as scripted TBM launches for their specific times. The steps performed by the initialization routine are shown in Figure 2.0-4.

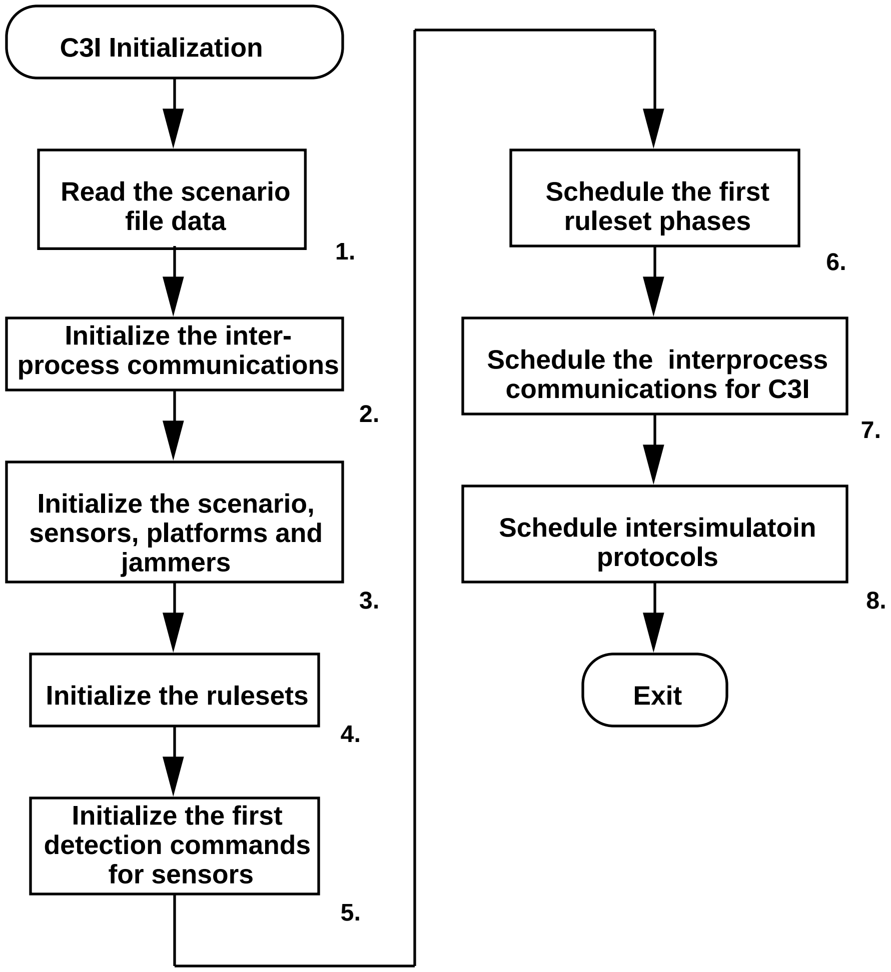

C3I Initialization
Read the scenario file data
1.
Initialize the inter-process communications
2.
Initialize the scenario, sensors, platforms and jammers
3.
Initialize the rulesets
4.
Initialize the first detection commands for sensors
5.
Schedule the first ruleset phases
6.
Schedule the interprocess communications for C3I
7.
Schedule intersimulatoin protocols
8.
Exit

FIGURE 2.0-4. C3I Process Initialization.

Block 1. The scenario file to be executed is read from disk in preparation for initializing the C3I platform data structures.

Block 2. The sockets between C3I and the other processes are initialized to establish communications with the other processes.

Block 3. The C3I data structures for all of the simulation objects, e.g., the scenario, platforms, sensors, weapon P_k tables and jammers, are initialized from the scenario data base.

Block 4. The rulesets phases are assigned to platforms as determined from the scenario data base.

Block 5. The initial commands for controlling platform sensors are constructed from the scenario data base and placed in the interprocess communications queue for Detection.

Block 6. The first ruleset phases to be executed for the platforms are placed on the calendar in time order.

Block 7. The C3I data transfer function is placed on the calendar as the first event. This order of events assures that the data transfers between C3I and the other processes occur in the sequence necessary to maintain the proper timing relationships between the processes.

Block 8. If EADSIM is communicating with other simulations via DIS or ALSP, the update function(s) for the specified protocol is scheduled on the calendar for execution.

C3I Interprocess Communications Logic

Figure 2.0-5 is a logic flow diagram of C3I Data Transfer which consists of a sequence of tests to determine if any external simulations are communicating with EADSIM and which of the other three processes are running in the current EADSIM configuration. Each read at a socket interface will block the C3I process until the expected data arrives and has been retrieved.

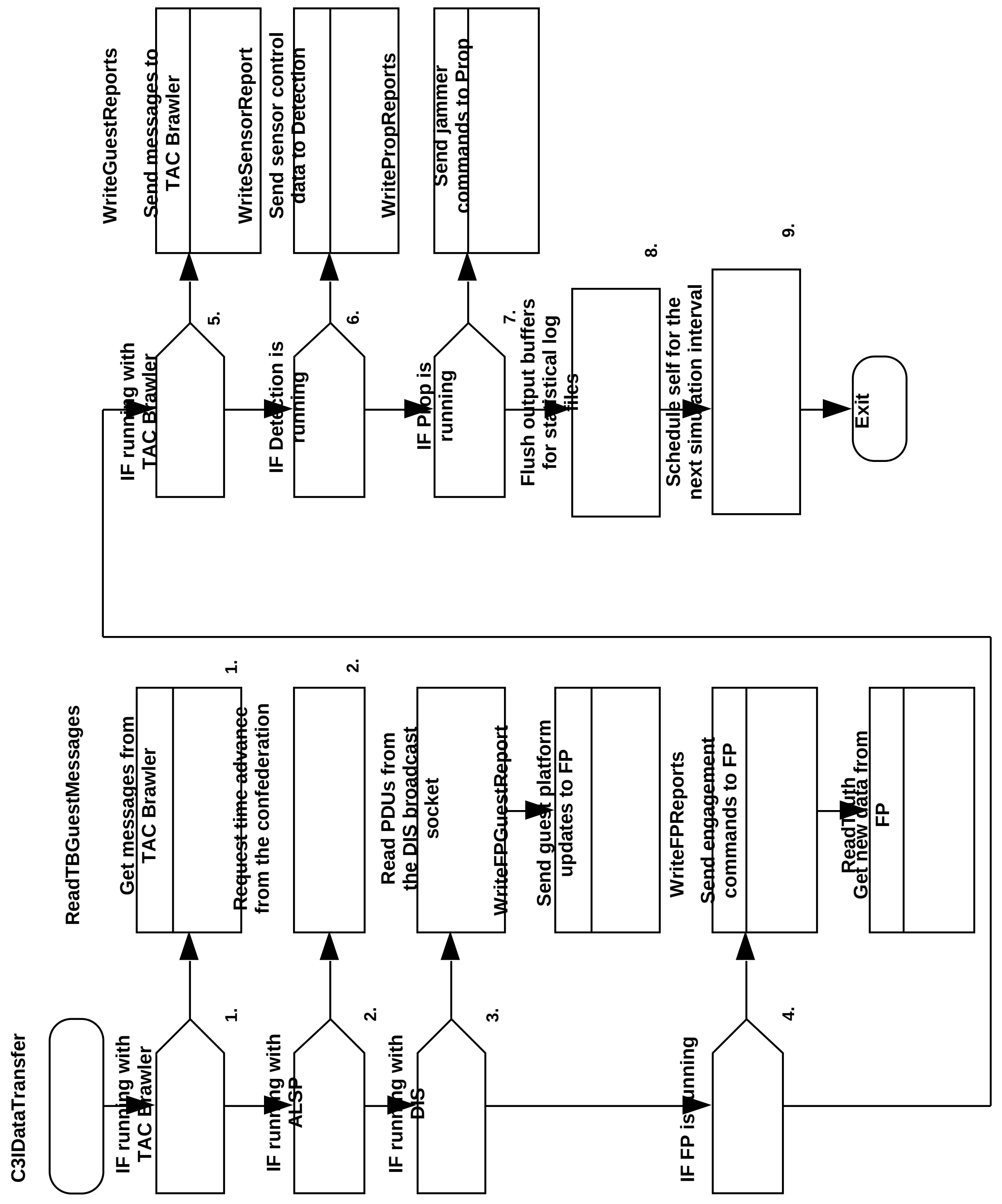

C3IDataTransfer
IF running with TAC Brawler
1.
ReadTBGuestMessages
Get messages from TAC Brawler
Request time advance from the confederation
1.
IF running with ALSP
2.
2.
IF running with DIS
3.
Read PDUs from the DIS broadcast socket
WriteFPGuestReport
Send guest platform updates to FP
IF FP is running
4.
WriteFPReports
Send engagement commands to FP
ReadTruth
Get new data from FP
IF running with TAC Brawler
5.
WriteGuestReports
Send messages to TAC Brawler
IF Detection is running
6.
WriteSensorReport
Send sensor control data to Detection
IF Prop is running
7.
WritePropReports
Send jammer commands to Prop
Flush output buffers for statistical log files
8.
Schedule self for the next simulation interval
9.
Exit

FIGURE 2.0-5 C3I Interprocess Communications.

Block 1. If TAC Brawler is running in confederation with EADSIM, the call to ReadTBGuestMessage passes incoming messages to C3I.

Block 2. If EADSIM is running with an ALSP confederation, a Request Time Advance message is sent out to coordinate EADSIM's time advance with the other simulations. EADSIM is forced to wait here until the other simulations in the confederation have advanced their time to or past EADSIM's requested time.

Block 3. If EADSIM is running with a DIS confederation (other than TAC Brawler), incoming messages are read from the DIS broadcast socket and the call to WriteFPGuestReports sends guest platform update messages to Flight Processing (guest platforms are platforms that are represented in EADSIM but are updated by another simulation).

Block 4. If Flight Processing is running (it's always required), commands from C3I to FP, generated in the previous simulation interval, are sent by the call to WriteFPReports. The following call to ReadTruth reads platform states, updated to the current simulation interval, and status information from FP and places it in the appropriate C3I platform data structure specified in each message.

Block 5. If EADSIM is running in confederation with TAC Brawler, messages generated in the current simulation interval are transmitted to TAC Brawler by the call to WriteGuestReports.

Block 6. If the Detection process is part of the current configuration, WriteSensorReports is called to send it sensor commands for the current interval. ReadSPDS reads the detections for the interval. ReadSPDS uses a C language function pointer in the platform data structure to call a 'plug in' function that reads the detection summary for the current simulation interval and performs the ruleset's track processing. This function is assigned to the ruleset during initialization. It uses the detection information to update the platform track files specified by each message and can purge a track to accommodate one of higher quality when the track file has no room for more tracks.

Block 7. If Propagation is running, the (receiver) jammer commands C3I has generated are sent for processing by calling WritePropReports. The call to GetCommunications reads network connectivity results computed by Propagation and places them in the designated network data structures.

Block 8. After data is received from the Propagation process, the output buffers for logging statistics are flushed.

Block 9. C3IDataTransfer reschedules itself to execute at the start of the next interval.

The executive now proceeds to call all events that have been scheduled for the interval. This process of scheduling C3IDataTransfer and subsequent calls to scheduled rulesets continues until the scenario ends.

Ruleset Execution

Figure 2.0-6 is a conceptual representation of the processing performed in a ruleset. The sequence of processing shown in the figure does not correspond closely to the actual processing in a ruleset; for example, a phase may be scheduled anywhere in a given ruleset. Typically, it is one of the last operations in a processing path. The figure also shows how information from the other processes is stored and accessed by the components of C3I. In addition to the data transfer function which comprises one component, rulesets can be divided into three more components: Track Processing, Message Processing and BM/C3 processing. The Message Processing component consists of a transmit function, which is referenced implicitly in Block 2, and the receive message processing function that processes messages sent out over a communications network.

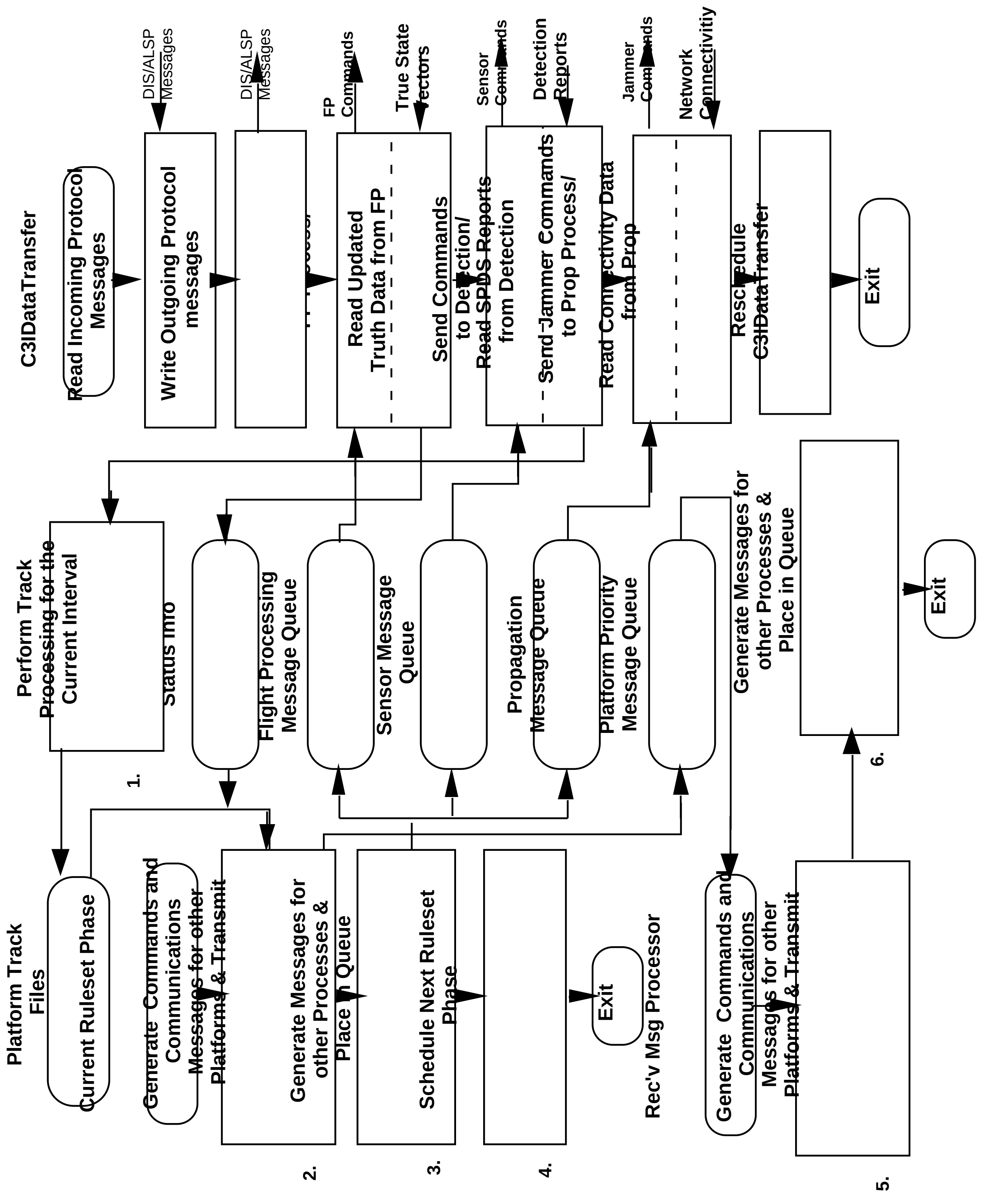

Platform Track Files
Current Ruleset Phase
Generate Commands and Communications Messages for other Platforms & Transmit
Generate Messages for other Processes & Place in Queue
Schedule Next Ruleset Phase
Exit
Rec'v Msg Processor
Generate Commands and Communications Messages for other Platforms & Transmit
1.
2.
3.
4.
5.
6.
Perform Track Processing for the Current Interval
Status Info
Flight Processing Message Queue
Sensor Message Queue
Propagation Message Queue
Platform Priority Message Queue
Generate Messages for other Processes & Place in Queue
Exit
C3IDataTransfer
Read Incoming Protocol Messages
Write Outgoing Protocol messages
Read Updated Truth Data from FP
Send Commands to Detection/
Read SPDS Reports from Detection
Send Jammer Commands to Prop Process/
Read Connectivity Data from Prop
Reschedule C3IDataTransfer
Exit
DIS/ALSP Messages
DIS/ALSP Messages
FP Commands
True State Vectors
Sensor Commands
Detection Reports
Jammer Commands
Network Connectivitiy

FIGURE 2.0-6. Ruleset Processing.

Block 1. Track Processing is performed at the time detection reports are retrieved from the Detection socket. For each track file active tracks are updated. Low priority tracks are deleted from saturated track files to make room for new, higher priority tracks. When a platform's track file is empty and it receives a new track, phase 1 (e.g., Target Select) of its ruleset is scheduled for execution after a track establishment delay.

Block 2. The current ruleset phase of a platform uses information in the platform's track file and the updated platform states (for the current platform, its targets and its subordinates) to control sensor status, determine platform movement and maneuvers, to control communications networks, etc. Commands to control subordinates and messages reporting status to commanders and peers are generated and placed in the transmission queue for the appropriate network links. The receive message processor is scheduled to distribute the messages to destination platforms if it is not already running.

Block 3. Messages for the other processes are generated to relay engagement decisions to the other processes and are placed in the FP, Detection and Propagation output buffers for transmission over their respective sockets.

Block 4. The next ruleset phase for the current platform is scheduled.

Flight Process Logic

Flight Process logic is shown in Figure 2.0-7. The process is time stepped and synchronizes with the other processes through blocking sockets that communicate in the manner described for C3IDataTransfer.

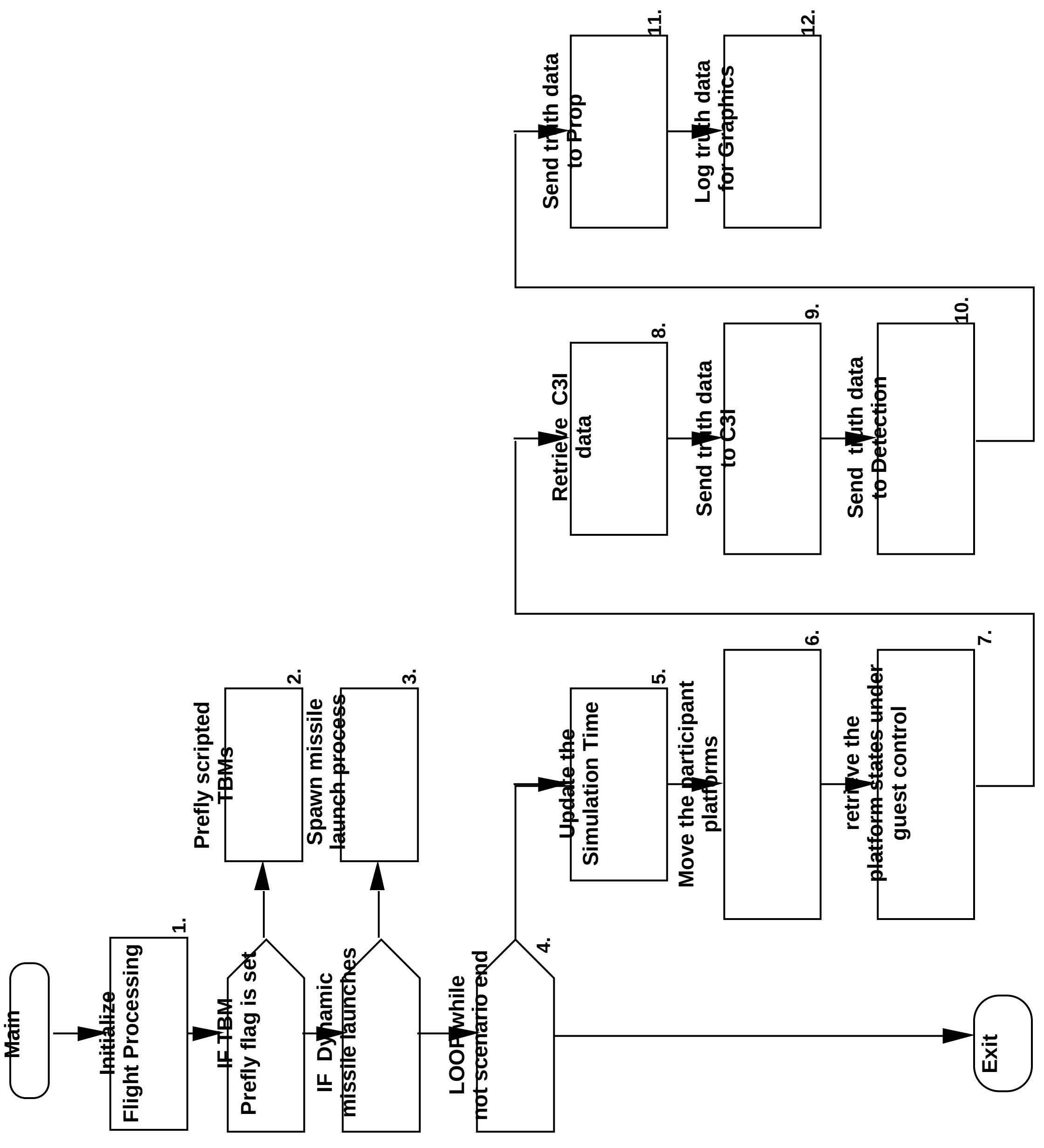


FIGURE 2.0-7. Flight Processing Logic.

Block 1. The scenario random number seed is initialized, the current Monte Carlo run number is updated and the number of Monte Carlo runs to be made is initialized. The scenario file is opened and the scenario database is used to initialize the FP data structures. The FP sockets for interprocess communications are initialized. The data structures for the Tactical Ballistic Missiles in the scenario are initialized. The list of weapons for each platform is built, waypoints lists are built and platform flight data structures are built. The graphics output file is initialized with static header information to allow state vector and status information to be logged in condensed form for playback.

Block 2. If the option has been selected, scripted TBMs are ‘preflown’ to compute the launch parameter values that will result in the prescribed trajectories, impact points and impact times.

Block 3. TBM launches can be optionally performed in a separate process if dynamically determined launches are required in a scenario. This block terminates FP initialization.

Block 4. The FP process remains in this loop until the end of the scenario.

Block 5. The FP process simulation clock is incremented to the current simulation interval prior to any processing in the interval.

Block 6. All active platforms and Tactical Ballistic Missiles (TBMs) are moved forward to the current interval. The specific movement algorithm used to move a platform is determined by the current mode of movement of the platform.

Block 7. If EADSIM is running in a DIS or ALSP confederation, platform update messages are read and used to update those platforms owned by other simulations in the confederation and represented in EADSIM. FP will remain blocked here until the expected information arrives.

Block 8. Commands and status information from C3I are read and used to determine which platforms are to be moved in the next simulation interval, what maneuvers are to be calculated, the mode of movement for each platform, the type of platform, etc. FP will remain blocked here until the expected information arrives.

Block 9. The true state vector and status information for each platform computed in Block 6 is sent to C3I where it will be used in engagement processing and to generate commands for the next simulation interval.

Block 10. The true state vector and status information for each platform computed in Block 6 is sent to Detection to generate detections for the current simulation interval.

Block 11. The true state vector and status information for each platform computed in Block 6 is sent to Propagation where it is used in determining network connectivity for the current simulation interval.

Block 12. The true state vector and status information for each platform computed in Block 6 is written to the graphics file for playback processing.

Control now moves to the top of the loop, through Blocks 5 and 6 and stops at Block 7 or 8 to wait for messages in the next simulation interval.

Detection Process Logic

Detection Process logic is shown in Figure 2.0-8. The process is time stepped and synchronizes with the other processes through the same type of blocking sockets described in section 2.1.1. The Detection process accepts commands from C3I and state vectors from FP to generate detections for sensors in the scenario each simulation interval. Detection’s flow of control is similar to that for FP in that it has an initialization step that precedes a processing loop. Once the processing loop is entered, it continues to be executed until the end of the scenario.

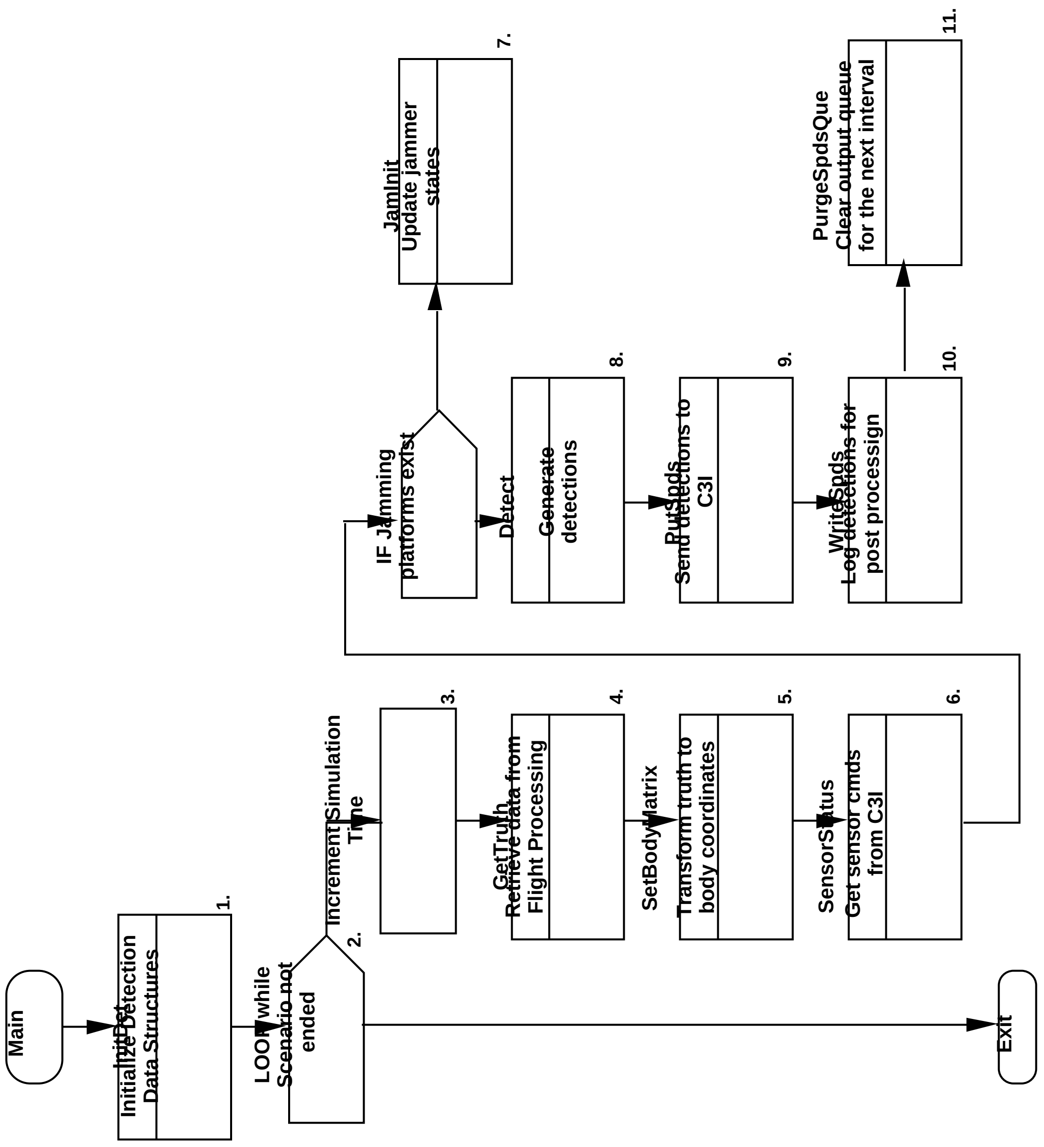

Main
1.
InitDet
Initialize Detection Data Structures
2.
LOOP while Scenario not ended
3.
Increment Simulation Time
4.
GetTruth
Retrieve data from Flight Processing
5.
SetBodyMatrix
Transform truth to body coordinates
6.
SensorStatus
Get sensor cmds from C3I
IF Jamming platforms exist
7.
JamInit
Update jammer states
8.
Detect
Generate detections
9.
PutSpds
Send detections to C3I
10.
WriteSpds
Log detections for post processign
11.
PurgeSpdsQue
Clear output queue for the next interval
Exit

FIGURE 2.0-8. Detection Process Logic.

<u>Block 1.</u> InitDet reads scenario data from the scenario data base, initializes the Detection interprocess communications sockets, the interface with the ALARM radar clutter and multipath models, sensor data structures and the Detection random number seed. It initializes the weapons and jammer data structures for each platform and the sensors for each platform.

<u>Block 2.</u> This is the top of the scenario processing loop for the detection process. Detection executes this loop for the duration of the scenario.

<u>Block 3.</u> The Detection simulation clock is advanced to the current simulation interval.

Block 4. Updated true state vectors and status information for each platform are read from the FP/Detection socket for the simulation interval. The process will wait here until all the expected data has arrived at the socket.

Block 5. SetBodyMatrix computes the transformation matrix to convert vectors from Earth Centered Inertial (ECI) coordinates to platform or missile body coordinates. The matrix is stored for later computations.

Block 6. Sensor commands from C3I for the current interval are read from the socket and the geometry, status and mode of each sensor on each platform in the scenario are updated. Sensor target lists are updated. The process will wait here until all the expected data has arrived at the socket.

Block 7. If the scenario contains jammers, the pointing angles are updated and converted from ECI to platform body coordinates according to commands received from C3I. Jammer status is also updated.

Block 8. Detect checks all the sensors on each platform in the scenario to determine those that can scan for targets. For each such sensor it determines which targets are detected and places the detection reports in a Sensor Periodic Detection Summary (SPDS) data structure. All the detections for the current simulation interval are calculated in this call to Detect.

Block 9. The SPDS data generated in Block 8. is sent over the Detection to C3I socket to C3I.

Block 10. Detection information is logged for post-processing.

Block 11. The SPDS queue is emptied.

Propagation Process Logic

Propagation is the third of the EADSIM time-stepped processes. It uses true platform state vectors and status information from FP and commands from C3I to determine the connectivity status of network links for each simulation interval. Figure 2.0-9 is a logical flow diagram of the main program for the Propagation process. It consists of initialization steps followed by a processing loop that lasts for the duration of the scenario.

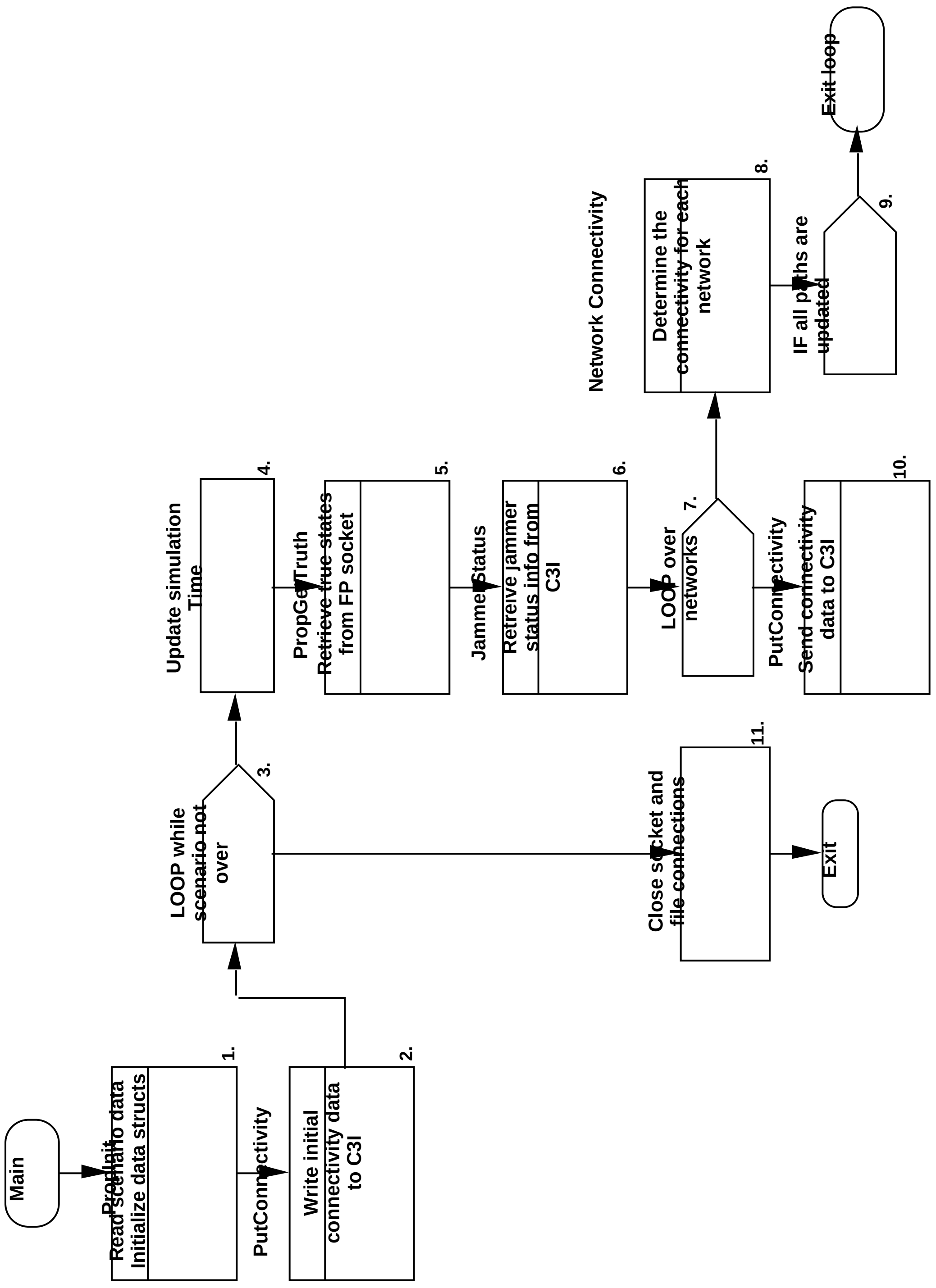

Main
PropInit
Read scenario data
Initialize data structs
1.
PutConnectivity
Write initial connectivity data to C3I
2.
LOOP while scenario not over
3.
Update simulation Time
4.
PropGetTruth
Retreive true states from FP socket
5.
JammerStatus
Retreive jammer status info from C3I
6.
LOOP over networks
7.
Network Connectivity
Determine the connectivity for each network
8.
IF all paths are updated
9.
Exit loop
PutConnectivity
Send connectivity data to C3I
10.
Close socket and file connections
11.
Exit

FIGURE 2.0-9. Propagation Process Logic.

Block 1. PropInit initializes the scenario data structures for the Propagation process and initializes the interprocess communications sockets with the other processes. It initializes the antenna locations and determines the antenna height for each network node.

Block 2. The initial connectivity data for all network links in the scenario is sent to C3I.

Block 3. The processing loop for Propagation executes for the duration of the scenario.

Block 4. The Propagation simulation clock is updated to the current simulation interval.

Block 5. The updated true state vectors from FP are read from the FP to Propagation socket. This information is used to update the position of each communications node that has moved in the last interval. The antenna altitudes are updated for communications nodes and jammers as are the heading and pitch angles for airborne antennas.

Block 6. Jammer commands from C3I are used to change the status of jammers and update jammer power. The on/off times for jammers is updated

Block 7. Loop over all the networks in the scenario.

Block 8. NetworkConnectivity performs connectivity analysis for each communications path of the given network. Paths are either connected or not.

Block 9. If the last path in the network to be updated just been processed, then control exits the processing loop. To enhance the execution speed, the links updated in a single interval can be specified as a percentage of the total.

Block 10. The connectivity information for the scenario networks in the current simulation interval is sent to C3I

Block 11. At the end of the current run the process gracefully shuts down.

Data Flow through the Major Components

EADSIM is comprised of three components: a preprocessing step, the runtime software, and a postprocessing step. The largest part of the preprocessing step is Scenario Generation, a graphical user interface through which the user defines the components and the layout of the scenario, as well as the initial sequence of events that start the scenario. The collected information is stored in a set of scenario files as shown in Figure 2.0-10 which are read by the runtime software and used to generate the scenario results. The results are recorded in the course of runtime software execution and stored in a set of data collection files which are read by the post processors to create analysis reports to the user and by the playback processor to generate an animated display of the scenario unfolding.

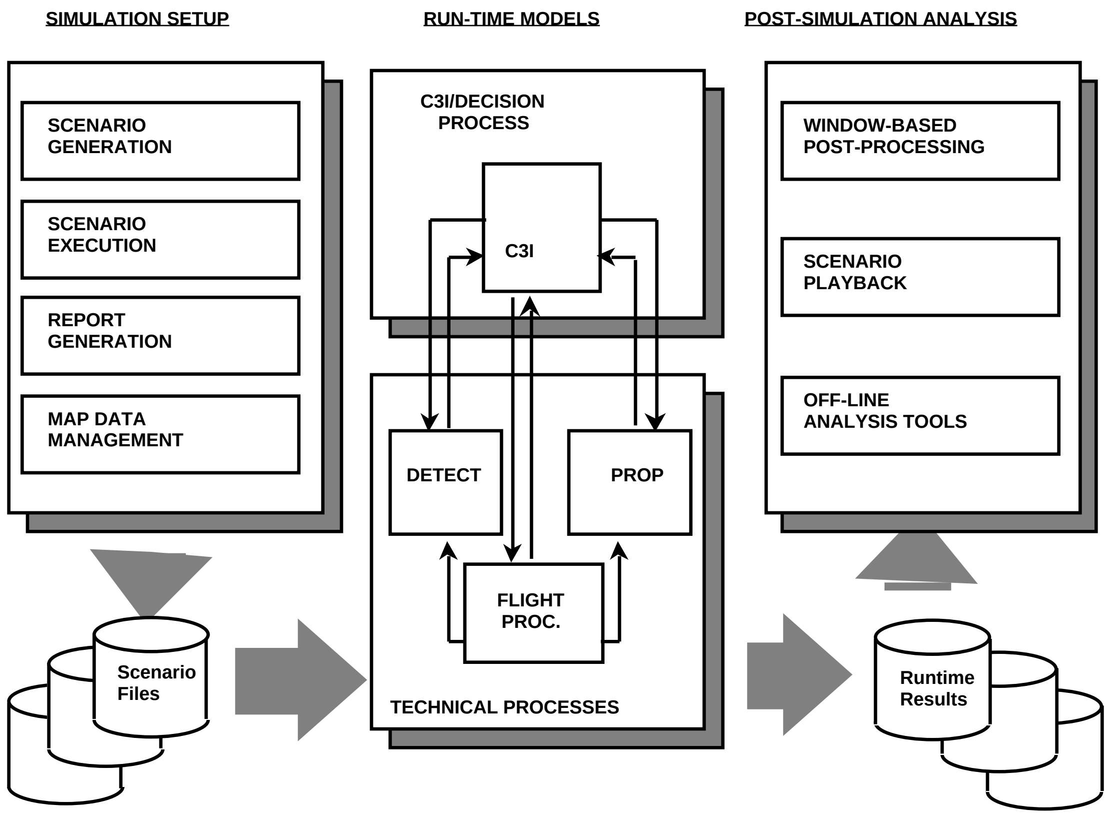


FIGURE 2.0-10. EADSIM Data Flow.

The scenario files contain information that characterizes the components that make up the scenario: the laydown files, platforms that comprise the laydowns, the systems that define the platforms and the elements that comprise the systems. Figure 2.0-11 illustrates the makeup of the scenario data. The data in a scenario is organized in a hierarchical fashion, with each level of the hierarchy building on the lower levels. The elements form the lowest level of the hierarchy:

Airframes	Protocols
Sensors	Radar Cross Section (RCS)
Rulesets	InFrared Signature (IR)
Communication Devices	Probability of Kill Tables (PK)
Jammers	Formation
Weapons	Flyout Table (FOT)

Combinations of elements are used to build Systems. Systems are deployed as Platforms. Groupings of Platforms are organized into Laydowns. The Platforms in the Laydowns are interconnected with Networks. The Networks also use the Protocol elements. Areas of Interest (AOIs) can be created and associated with both Platforms and Networks. The Map specification forms the geographic basis for the scenario. The Scenario combines all of the lower level data.

Each of these groupings of data is combined into files and directories. The element data is contained in a single directory, referenced as an element data path in the scenario. Each laydown is in a separate laydown file. Each grouping of networks is in a network file, and AOIs are in AOI files. The map specification is in a map file and is itself a specification of lower level data. LLTRs, Preferences, Display Colors and Routes, Transmittance and Radiance are likewise contained in separate files. Finally each scenario is contained in a scenario file.

SCENARIO GENERATION DATA HIERARCHY

A HIERARCHY OF DATA ORGANIZES AND EXPEDITES THE SPECIFICATION OF A SCENARIO

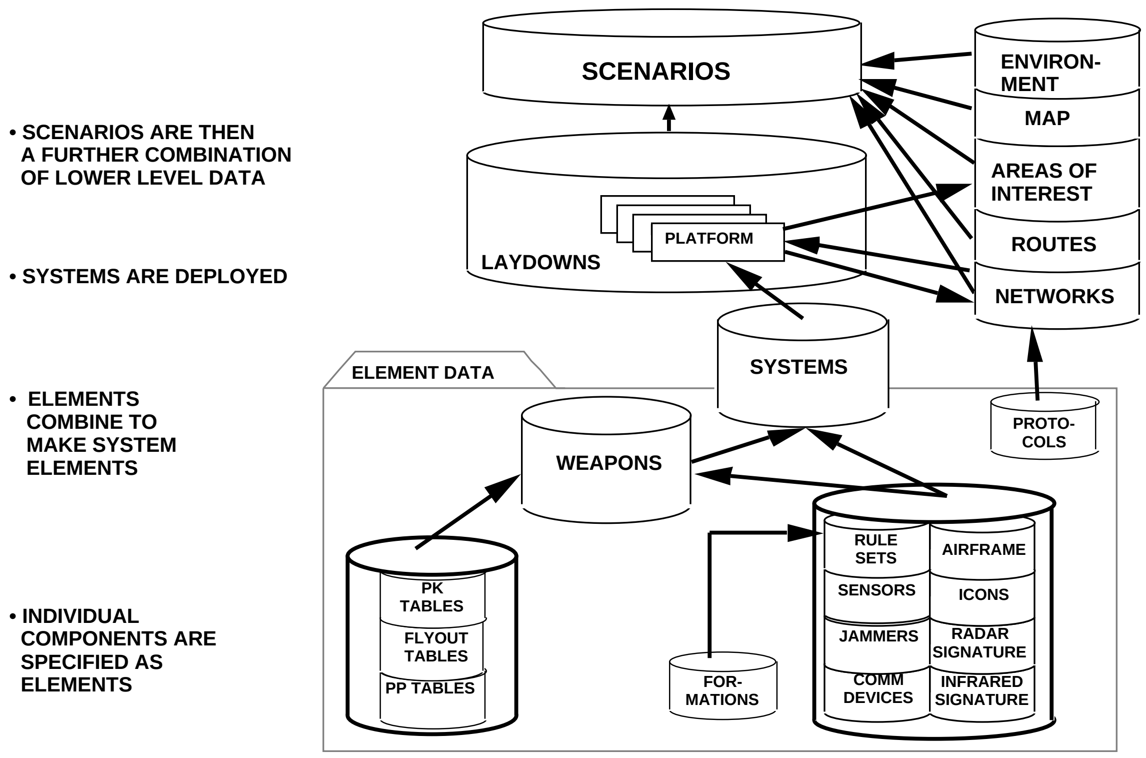


FIGURE 2.0-11. Scenario Database.

At each level of hierarchy (Figure 2.0-11) a file contains either direct references to pieces of the lower level files or path names to the lower level files. This feature of the input files facilitates a great deal of flexibility in scenario design and analyses. Major pieces of the data base can be ‘mixed and matched’ to form variants on scenarios without rebuilding the entire scenario. Also, each level can be used in multiple scenarios. Thus, once a reliable element database has been constructed, it can be reused in multiple scenarios. Different combinations of laydowns, networks, and maps can also be made. The contents of specific portions of the scenario file are read at the initialization of each of the four processes to assign initial values to the internal data structures of each process.

The internal data flow between processes is shown in Figure 2.0-12.

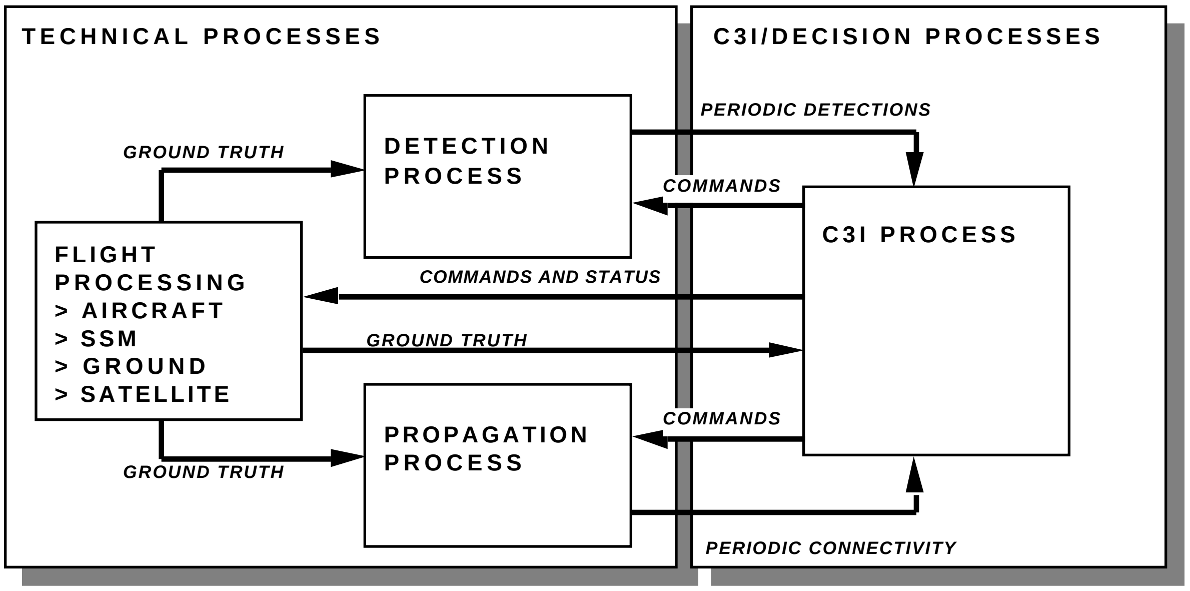


FIGURE 2.0-12. Runtime Process Data Flow.

Data is transferred between the processes in the form of messages transmitted over sockets. Sockets are special file descriptors that allow two processes residing on the same or different computers to communicate. Flight Processing sends the updated state vector for the active platforms in a scenario to Detection, Propagation and C3I. The Detection process passes detections to C3I every simulation interval in the form of a SPD Summary. Similarly, Propagation sends periodic connectivity reports to C3I. C3I sends commands and status information to Flight Processing. It sends commands to Detection that govern how sensors are used and to Propagation to control communications devices.

The output files contain logging information collected at runtime that is used by a post-processor to generate analysis reports and to drive the playback processor. The playback processor provides an animated display of the scenario as it was executed at runtime.

Source Code Hierarchy

Figures 2.0-13 through 2.0-16 are source code hierarchy charts for the four EADSIM runtime processes, C3I, Flight Processing, Detection and Propagation, respectively.

C3I Process

The calling tree for C3I is shown at the top of Figure 2.0-13 and consists of the call to the initialization function, C3IIinit and a call, via function pointer, to the next function scheduled for execution on the simulation event calendar. The lists headed by ‘*’ in the remainder of the figure contain functions that can be scheduled. This is not an exhaustive list; rather, it emphasizes the areas of interest for purposes of planned SMART analyses. The functions listed in the calling tree are implementations of the phases for the four rulesets, Flex SAM, Flex Commander, AG Attacker and Ground Attack Commander. A brief description of each function follows.

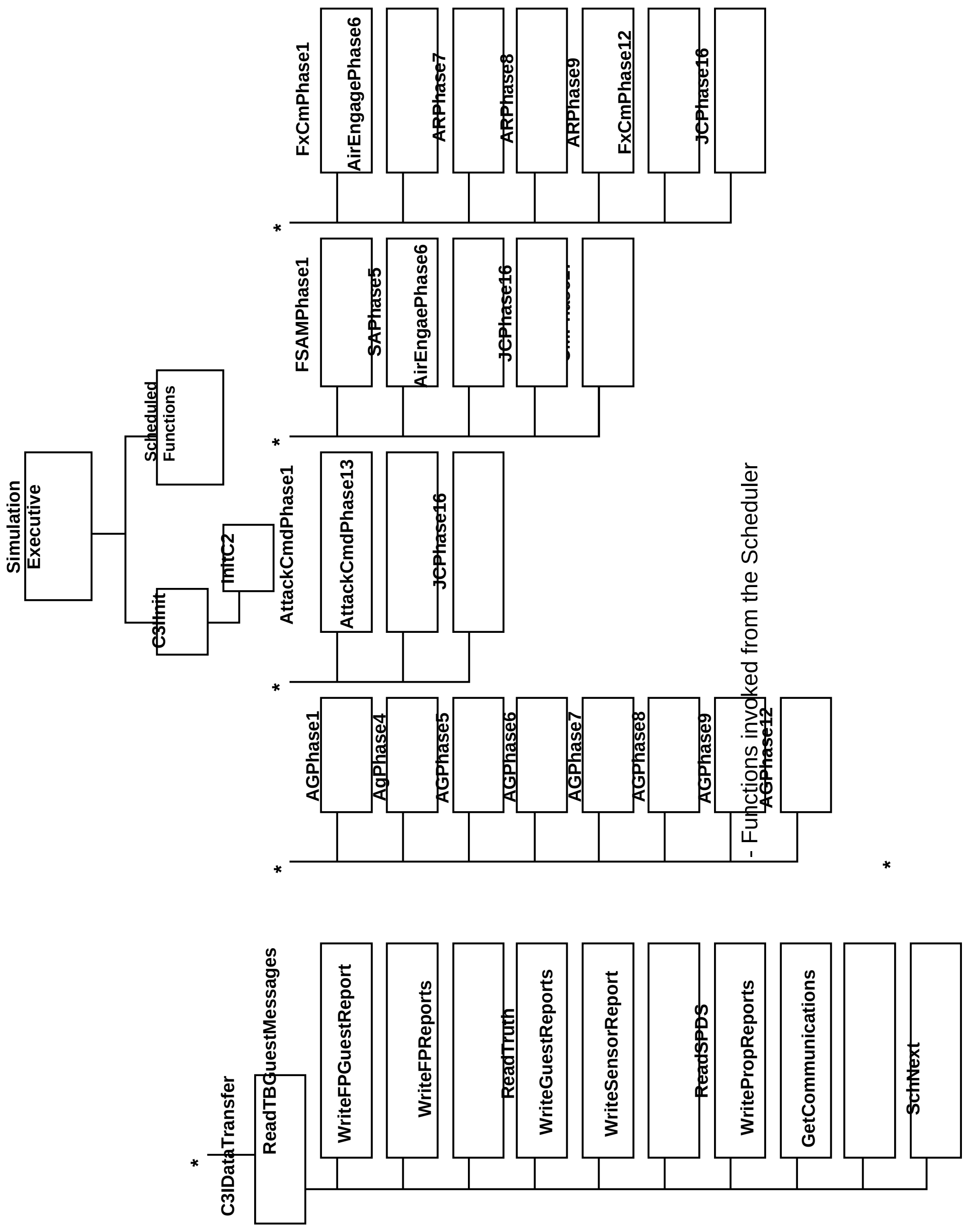


FIGURE 2.0-13. C3I Process Calling Tree.

Simulation Executive: The executive calls the C3I initialization function, performs calendar scheduling and places scheduled functions in execution.

C3IInit: Reads the scenario database and initializes the C3I runtime data structures. It generates the first detection commands for the sensors and schedules the initial events on the simulation calendar.

InitC2: Assigns runtime functions to the rulesets.

C3IDataTransfer: Runs at the beginning in each simulation interval and retrieves messages from the other processes that are required by C3I to perform its processing for the interval. Sends messages to control the other processes. Reschedules itself after completing data transfers for the current interval.

ReadTBGuestMessages: Reads DIS messages from TAC Brawler.

WriteFPGuestMessages: Sends Brawler messages to Flight Processing.

WriteFPReports: Sends engagement commands to Flight processing, e.g. determines maneuvers to be executed by a platform.

ReadTruth: Reads state vectors and status information for each platform sent from the FP process. The information has been updated by FP to be valid in the current simulation interval.

WriteGuestReports: Sends C2 information and state data to TAC Brawler.

WriteSensorReport: Sends sensor control and status messages for the current interval to the Detection process.

ReadSPDS: Reads detection reports from the Detection process for the current interval. C3I uses the information to update track files for each platform.

WritePropReports: Sends commands to update jammer information and dynamically establish new networks.

GetCommunications: Reads messages from the Propagation process and uses the information to determine whether network communications can occur.

SchNext: Schedules a call to C3IDataTransfer at the start of the next interval.

The AGAttacker ruleset phases are:

AGPhase1: Target select phase for the air-to-ground attacker ruleset. Executes for wingman for scripted targets and commanded targets from the flight leader. Executes for flight leaders for scripted, detected and commanded targets**.**

AGPhase4: Lock phase for the air-to-ground attacker ruleset. Executes until the platform's weapons are within launch range.

AGPhase5: AGAttacker launch phase. Executes once and schedules the intercept at weapon impact.

AGPhase6: Intercept phase for the air-to-ground attacker performs kill assessment on the target at intercept time.

AGPhase7: React to air-to-air engagement for the air-to-ground attacker. Executes when an AG Attacker is under attack and the attacker (fighter) is in engage mode. Determines the AG Attacker's reaction.

AGPhase8: AG Attacker reacts to lock by air attacker when the attacker has locked a weapon on the air-to-ground attacker.

AGPhase9: Air-to-ground attacker drag phase. Scheduled when a decision is reached that the AGAttacker should execute a drag maneuver.

AGPhase12: AGAttacker reacts to SAM lock. The SAM platform schedules AGPhase12 when it locks on the AGAttacker platform, providing that the SAM has a Radar sensor.

JCPhase16: Allocates jammers on a target platform (e.g. AG Attacker) to jam a threat radar. This function executes periodically.

The Ground Attack Commander ruleset phases are:

AttackCmdPhase1: Performs the target select phase for the Ground Attacker Commander.

AttackCmdPhase13: Performs the vector update for a Ground Attacker Commander, i.e., a commanded AG Attacker is informed that a commanded intercept point has been changed.

JCPhase16: Allocates jammers on a target platform (e.g. AG Attacker) to jam a threat radar. This function executes periodically.

The Flexible SAM ruleset phases are:

FSAMPhase1: Performs target selection and weapons assignment for flexible SAM platforms.

SAPhase5: Executes at the time of launch for a ground based weapon system. If the launch has not been canceled, it will begin the engagement and schedule a call to the intercept routine at intercept time. Otherwise it will return without doing anything.

JCPhase16: Allocates jammers on a target platform (e.g. AG Attacker) to jam a threat radar. This function executes periodically.

CMPhase17: Countermeasures phase, currently used for Flexible SAMs and Flexible Commanders to call a trigger evaluation function that determines if circumstances indicate a reaction. Calls a response routine to execute the reaction.

The Flexible Commander ruleset phases are:

FxCmPhase1: Target select phase for a Flexible Commander. This phase contains the logic that assigns both air and ground based assets to targets. These assets include fighters, SAM systems and DEW systems. The Flex Commander can be assigned to an airborne or a ground based platform.

AirEngagePhase6: Executes at the intercept time for a ground or air based weapon system. If the intercept has not been canceled, it will determine if the intercept was successful and perform the status operations. Otherwise it will return without doing anything.

ARPhase7: Executes when a subordinate tanker is under attack and the attacking system is in engage mode. Schedules SCRAM drag maneuver for himself. Since the Flex Commander reactions to attack are similar to those of the Air Refueling (AR) Tanker, the ruleset uses the AR phases 7, 8 and 9 for this purpose.

ARPhase8: Executes when an air borne commander is under attack and the attacking system has locked a weapon. Schedules the SCRAM drag maneuver for himself.

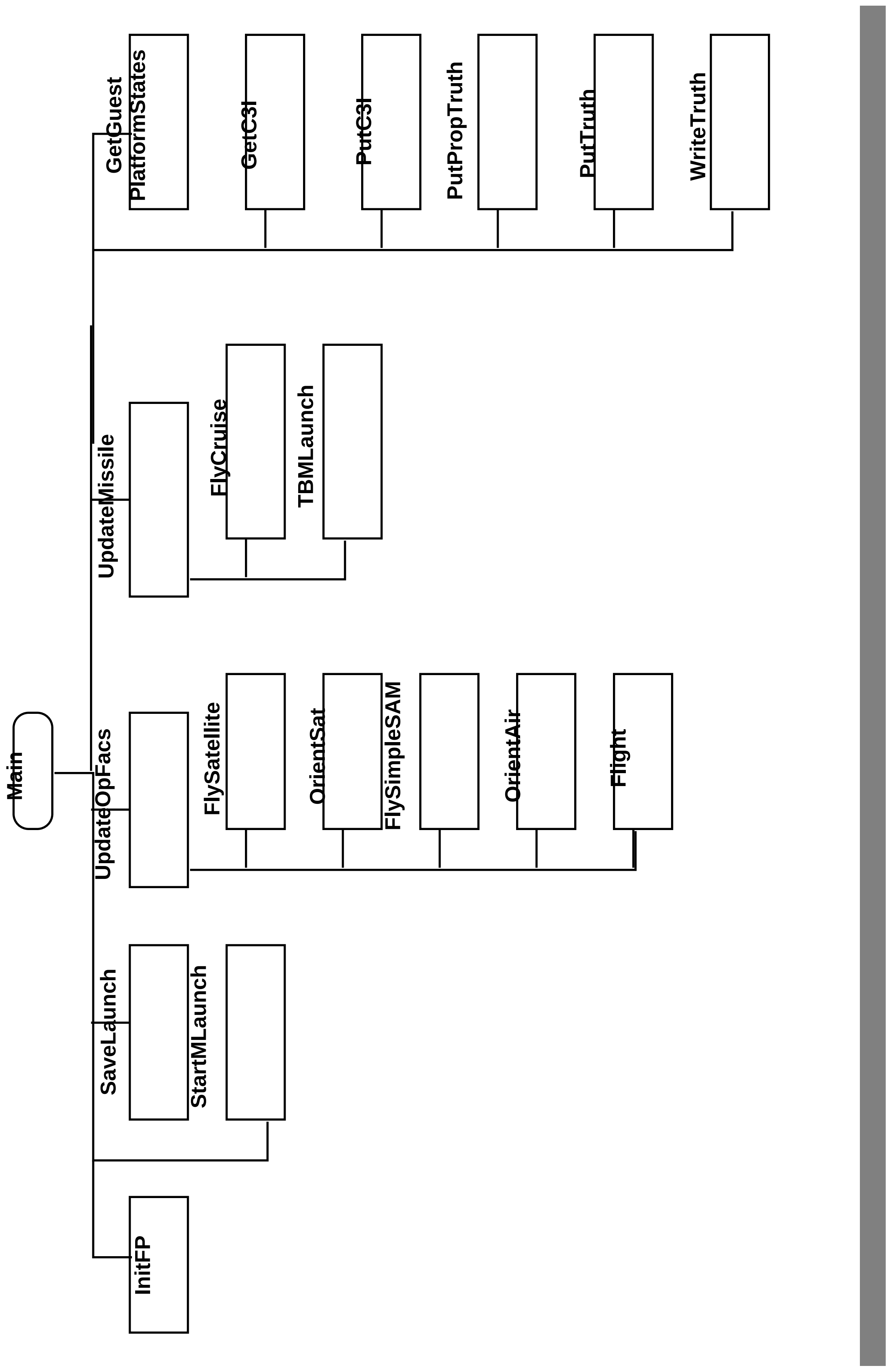
ARPhase9: Reaction routine is scheduled when a platform should execute a drag maneuver. Assumed to be called once the maneuver has begun to determine when it will complete.

FxCMPhase12: Executes when a SAM platform locks on an airborne flexible commander The SAM platform initially schedules FxCmPhase12 (providing that the SAM has a Radar sensor)

JCPhase16: Allocates jammers on a target platform (e.g. AG Attacker) to jam a SAM platform radar. This function executes periodically.

Flight Processing Process:

The calling tree for the FP process is shown in Figure 2.0-14. The functions performed in the FP process can be grouped into Initialization, TBM Launch, Platform Update, Missile Update and Interprocess Communications. The two update functions differ in that platform updates affect different parameters than do missiles. A brief description of each function follows.


Main
InitFP
SaveLaunch
StartMLaunch
UpdateOpFacs
FlySatellite
OrientSat
FlySimpleSAM
OrientAir
Flight
UpdateMissile
FlyCruise
TBMLaunch
GetGuest PlatformStates
GetC3I
PutC3I
PutPropTruth
PutTruth
WriteTruth

InitFP: Reads the scenario data for flight processing and initializes the FP data structures. It also establishes interprocess communications with the other processes.

SaveLaunch: Preflies the scripted TBMs in the scenario and saves the launch parameters for use during TBM flyout.

StartMLaunch: Spawns a separate process that is responsible for finding launch solutions for missiles that are launched dynamicallyi.e. in response to an event or command during scenario execution.

UpdateOpFacs: Updates the position and status of a platform for the current time step. Moves airborne, surface, and missile Platforms.

FlySatellite: Propagates the position and velocity of an object along a ballistic drag free elliptical, hyperbolic or parabolic trajectory using an F&G series integrator.

OrientSat: Orients the body of a satellite with respect to its velocity vector.

FlySimpleSAM: Performs 3 D.O.F. and Constant Velocity SAM flyouts. Models the guided flight of a homing missile from launch to time of target closest approach.

OrientAir: Orients the body frame of an airborne platform with respect to its velocity vector.

Flight: Calls the function that performs calculations for a specified flight mode, e.g., waypoint mode, scramble, return to base, wingman, etc.

UpdateMissile: When the scheduled launch time for a missile occurs, UpdateMissile performs the launch calculations and continues to periodically update its state until impact occurs or the missile is killed.

FlyCruise: Updates the state of a cruise missile between waypoints and maintains the missile's position within an altitude corridor or at a constant altitude above the ground.

TBMLaunch: Iterates on pitch angle or pitch rate to converge on a trajectory that will achieve the desired ground range and then initializes a TBM for launch.

GetGuestPlatformStates: Used by Flight Processing to retrieve the states of platforms owned by external simulations from the socket connection between FP and C3I.

GetC3I: This function receives engagement commands from the C3 mode that is part of this simulation run. It unpacks data from a buffer and then sets the flight processing mode and system state as indicated by the buffer contents.

PutC3I: Sends updated truth states and status information to the C3I process. Called every simulation interval.

PutPropTruth: Transmits the truth data from Flight Processing to Propagation.

PutTruth: Transmits the truth data from flight processing to sensor detection.

WriteTruth: Used by flight processing to send the truth data to a graphics display file. Data is logged only when a system is active or has been recently killed.

Detection Process

The Detection process calling tree is shown in Figure 2.0-15. Functional groupings are: Initialization, Interprocess Communications and functions which perform the detection computations. A brief description of each of the functions shown in the calling tree follows.

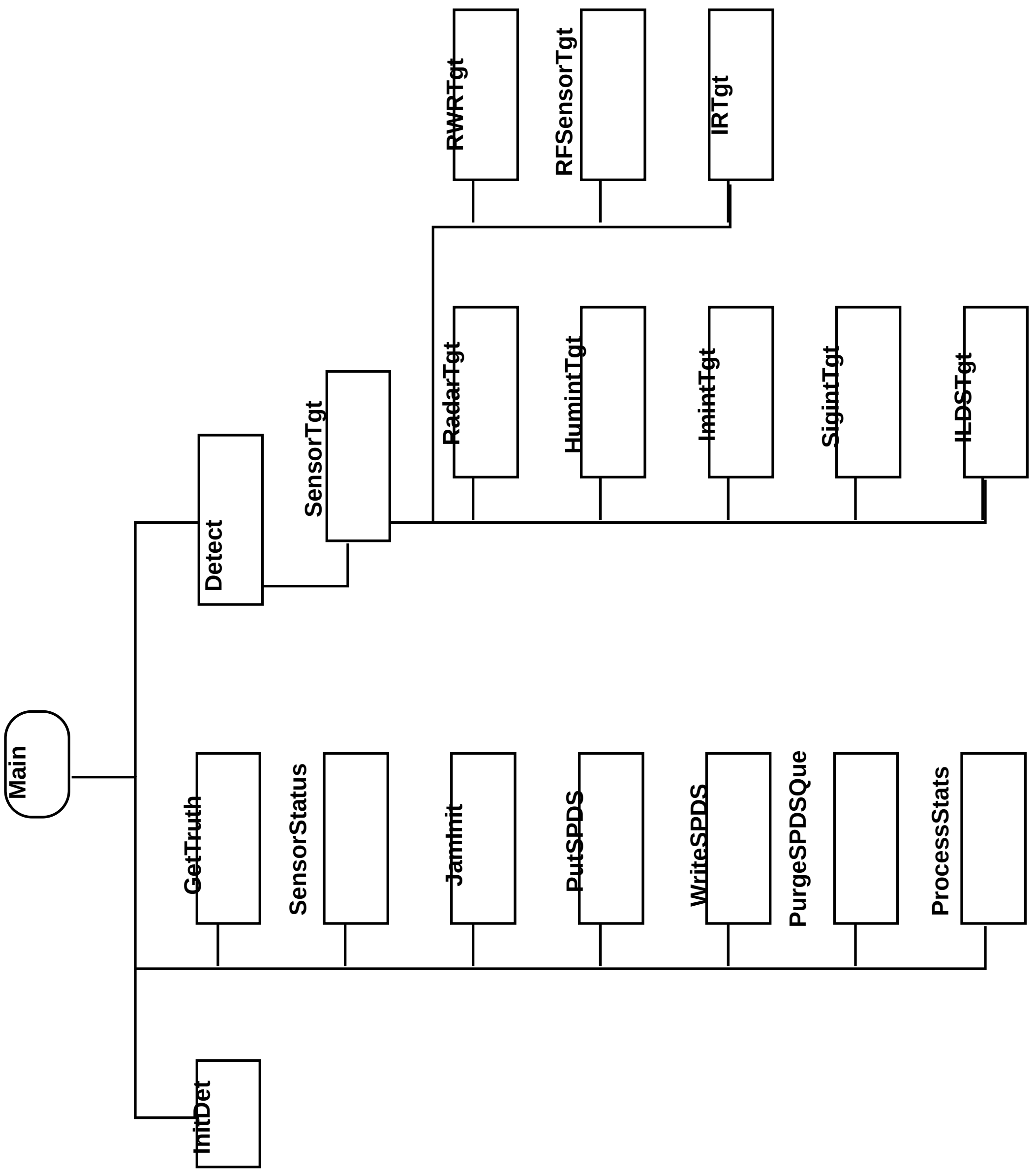


FIGURE 2.0-15. Detection Processing Calling Tree.

InitDet: Uses data retrieved from the scenario file to initialize the detection data structures and ancillary functions, e.g. selected ALARM subroutines used to support clutter and multipath calculations.

GetTruth: Used by sensor detection to retrieve truth states and status information sent from Flight Processing.

SensorStatus: Uses sensor commands from C3I to update sensors in preparation for generating the detections for the current interval.

JamInit: Sets up jammer pointing angles and transforms coordinates from ENU to the jammer reference frame.

PutSPDS: Sends sensor detections for the current simulation interval to the C3I process.

WriteSPDS: Logs detections for the current interval to the Detection data collection file.

PurgeSPDSQue: Clears the Sensor Periodic Detection Summary output buffer (queue) at the end of each simulation interval.

ProcessStats: Computes and logs Detection process statistics each 60 seconds of simulation time. These include CPU and memory usage.

Detect: Updates the status of each sensor in response to commands from C3I. Turns sensors on or off in the presence of jammers and ARMs, adjusts sensor point angles in preparation for computing detections

SensorTgt: Determines the type of sensor to model and calls the appropriate detection function. Serves as the gateway to the detection algorithms.

RadarTgt: Generates radar detections. Detection calculations are determined by user inputs and can include deterministic/probabilistic detections, clutter, multipath and diffraction, simple sensor, compound sensor, radar resource management, etc.

HumintTgt: Loops through all pertinent sublists to determine which target systems the human intelligence sensor can detect.

ImintTgt: loops through all pertinent systems to determine which target systems the specified image intelligence sensor can view

SigintTgt: Models a signal intelligence sensor. This type of sensor detects targets based on the type and frequency of active radio emitters on the target system.

ILDSTgt: This module loops through all launched missiles to determine if the specified infrared launch detector sensor can view which of the missiles.

RWRTgt: Loops through a list of systems to determine which of its scripted target systems the specified radar warning receiver can view.

RFSensorTgt: loops through all pertinent systems, checking the on/off times, to determine which of the target systems the input passive RF sensor can view.

IRTgt: loops through all pertinent sublists to determine which target systems the input infrared sensor can view.

Propagation Process

The Propagation process calling tree, shown in Figure 2.0-16, consists of an initialization function, functions for interprocess communications, jammer status updates and connectivity computations. A brief description of each of the functions shown in the calling tree follows.

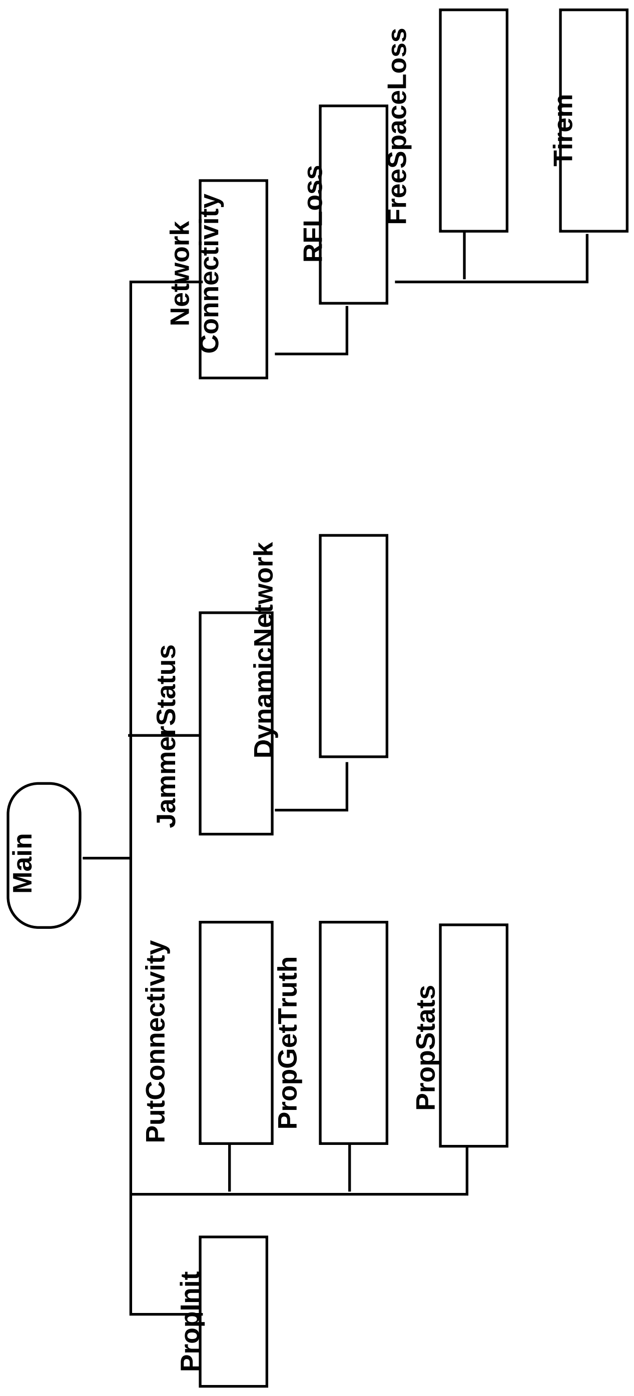


FIGURE 2.0-16. Propagation Process Calling Tree.

PropInit: Reads the scenario file and initializes the propagation data structures.

PutConnectivity: Used by propagation to transmit the connectivity matrix to C3I. The matrix shows which of the defined network links in the scenario have connectivity in the current scenario interval. Matrix values are 0, for no connectivity, or 1, if connected.

PropGetTruth: Retrieves platform truth states and status information sent from Flight Processing.

PropStatus: Used by propagation to write the connectivity stats to the stats file.

JammerStatus: Changes jammer status in response to commands from the C3I process. Changes include turning the jammer on and off, dynamically allocating jammers, etc.

DynamicNetwork: Allocates memory for the explicit network of a dynamically allocated platform.

NetworkConnectivity: Performs connectivity analysis for each communications path of the specified network.

RFLoss: Determines whether the link between two network nodes has unobstructed line of sight. If so, it uses free space loss calculations to determine connectivity. Otherwise it uses the TIREM model to make the determination.

FreeSpaceLoss: Computes the free space loss in db.

TIREM: Determines connectivity in the presence of obstructions in the line of sight between two nodes.